Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Final prototype of LinuxSSI
D2.2.11

Due date of deliverable: March 31st, 2010
Actual submission date: May 7th, 2010

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP2.2
Task number: T2.2.3,T2.2.6,T2.2.9,T2.2.10,T2.2.11,T2.2.12

Responsible institution: XLAB
Editor & and editor's address: Marko Novak
XLAB
Pot za Brdom 100
1000 Ljubljana
Slovenia

Version 1.5 / Last edited by Marko Novak / May 12, 2010

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	22/02/10	Marko Novak	XLAB	Prepaired the structure for the deliverable.
0.2	11/03/10	Matthieu Fertré	KER	Write the checkpointing section.
0.3	31/03/10	Jean Parpaillon	KER	Write the test, integration and support section.
1.0	19/04/10	Marko Novak	XLAB	Finished the deliverable.
1.1	21/04/10	Matthieu Fertré	KER	Adding comments for shell commands
1.2	21/04/10	Matthieu Fertré	KER	Rewriting section 2.2
1.3	21/04/10	Matthieu Fertré	KER	Adding more explanations about SHM checkpoint/restart
1.4	05/05/10	Peter Linnell	INRIA	Final Proofing, English Clarifcations
1.5	07/05/10	Marko Novak	XLAB	Integrated the last suggestions from reviewers into the deliverable.

Reviewers:
Toni Cortes (BSC) and Marko Obrovac (INRIA)

Tasks related to this deliverable:

Task No.	Task description	Partners involved[°]
T2.2.3	Design and implementation of advanced checkpoint/restart mechanisms	KER[*]
T2.2.6	Design and implementation of advanced scheduling policies	XLAB[*]
T2.2.9	Pushing SSI mechanisms into Linux mainstream development	NEC[*]
T2.2.10	Outgoing IP communication	KER[*]
T2.2.11	Dynamic streams with checkpoint/restart	KER[*]
T2.2.12	Test, integration and support	KER[*]

[°]This task list may not be equivalent to the list of partners contributing as authors to the deliverable
[*]Task leader

Executive Summary

This document is a final report on the XtreemOS cluster flavour prototype. It reports on the work done as part of WP2.2 during the final period (M37-M48) of XtreemOS cluster flavour development. LinuxSSI is the heart of the foundation layer for XtreemOS cluster flavour. LinuxSSI leverages Kerrighed single system image cluster operating system developed via open source (http://www.kerrighed.org).

The document describes checkpointing of processes which use inter-process communication (IPC) constructs. IPC constructs are used for exchanging data among two or more threads in one or more processes. System V IPC provides semaphore arrays, message queues and shared memory segments to manage communication among processes. At checkpoint time, it is impossible to automatically discover which IPC resources have been, and will be, used by the application the user want to checkpoint. Moreover, one IPC resource may be used to help collaboration between several applications. Since the Kerrighed checkpointer is application-based, the user has to choose which IPC resources to checkpoint when they checkpoint an application.

The document also describes dynamic streams with checkpoint/restart ability. In common cases, communicating processes running on a single machine may use protocols such as TCP/IP, UDP/IP, Unix sockets, System V queues or pipes. In a single system image like Kerrighed, we have to support such mechanisms that enables uninterrupted communication even after the processes have been migrated to a different cluster node. The quick, but inefficient way to provide this feature would be to forward the messages among the cluster nodes. The drawback of this method is the large overhead introduced regardless of the actual process location: processes running on the same node, must forward all their messages! Our goal was to allow efficient network communication between processes running in the same Kerrighed cluster. In our mind, "efficient" means little or residual performance overhead. our aim is avoiding message forwarding between cluster nodes when processes migrate.

LinuxSSI also has to provide support for TCP/IP communication between a process running inside a Kerrighed cluster and another running outside the cluster. Since the processes in SSI cluster migrate among different nodes, we need the mechanism to be able to communicate with them from the outside, regardless of the node on which they are currently residing. For example, one of our goals was to have a fully transparent mechanism at the protocol level to allow an outside client to access a TCP/IP server running inside a Kerrighed cluster. No matter on which node the TCP/IP server is running, the client must be able to communicate with it.

The next section of the deliverable describes the Scheduling Configuration Changer, the framework for reconfiguration of the probes and scheduling policies based on the current cluster load. These mechanisms add the capability to dynamically adjust varying computational demand. Furthermore,this scheduler exchange mechanism can be used to prioritize different goals at different times. i.e. A well thought out policy would migrate several interactive jobs that are idle during the night to a small subset of machines and switch off all other machines. This may lead to larger energy savings which impact a project's budget.

In addition to purely technical topics, the document also describes continuing efforts to push LinuxSSI to the Linux community. The motivation for this was to attempt to to move parts of LinuxSSI into the mainstream Linux Kernel. Similarly, we wanted to discover a path that would make the ideas and concepts behind LinuxSSI and Kerrighed more visible to the Linux Kernel developer community, along with finding users and use cases outside the LinuxSSI project that could supported integration

into the Linux Kernel.

The last section the LinuxSSI integration activities that were perform in the final period of the XtreemOS project. In this context, our first goal was to provide a large collection of tests which can be run manually by developers or users. These tests must ensure Kerrighed respects the Linux API and behaviour. The second goal was to setup an automated testing platform, which would enable to run these tests as many times as possible with as many configurations as possible.

Contents

1 Introduction

This document is the final report on the XtreemOS cluster flavour prototype. It reports on the work done as part of WP2.2 during the final period (M37-M48) of XtreemOS cluster flavour development.

LinuxSSI is the heart of the foundation layer for XtreemOS cluster flavour. LinuxSSI leverages Kerrighed single system image cluster operating system developed in open source (http://www.kerrighed.org). Most of our work as part of WP2.2 focuses on the design and implementation of new features in LinuxSSI.

In this document, we describe the main features that have been developed in XtreemOS cluster flavour since the June 2009. The new features relate to the following features:

- Application checkpoint/restart developed as part of Task T2.2.3 *Design and implementation of advanced checkpoint/restart mechanisms*,
- Adaptable cluster load balancer developed as part of Task T2.2.6 *Design and implementation of advanced scheduling policies*,
- IP communication adaptation for LinuxSSI developed as part of T2.2.10 *Outgoing IP communication*,
- Managing open streams when performing the checkpoint/restart developed as part of T2.2.11 *Dynamic stream with checkpoint/restart*.

Since M30 of the XtreemOS project, the implementation of LinuxSSI has been quite stable, nor has its interfaces significantly changed. The majority of the data provided in [**?**] and [**?**] is still relevant for the latest version of LinuxSSI. The reader that wishes to learn more details about using LinuxSSI should first check those documents.

The next chapter (2) describes checkpointing processes which use interprocess-communication (IPC) constructs. It explains how the checkpoint/restart for IPC objects operates from a user point of view and what are the commons in the implementation of checkpoint/restart for IPC resources. Chapter 5 describes Scheduling Configuration Changer. This is the framework for reconfiguring probes and scheduling policies based on the current cluster load. The system can automatically extend the probing period to reduce overhead due to instrumentation. When the system load is back to a more normal state, probing periods are adjusted again. Chapter 6 describes the ongoing effort to push LinuxSSI to the Linux community, to find users and use cases outside the LinuxSSI project that would support integration into the Linux Kernel. Chapters 4 and 3 describe a fully transparent mechanism at the network communication protocol level that enabling access to the TCP/IP servers running on LinuxSSI cluster, along with the checkpointing and migration of processes which use communication sockets. Testing activities are described in chapter 7. This chapter concerns the description of the testing framework.

2 Design and Implementation Of Advanced Checkpoint/restart Mechanisms

Inter-Process Communication (IPC) is a set of techniques for exchanging data among two or more threads in one or more processes. System V IPC provides semaphore arrays, message queues and shared memory segments to manage communication among processes in User Mode on one computer. They were introduced in Unix System V. Like files, IPC resources are persistent: they must be explicitly deallocated by the parent process, by the current owner, or by a superuser process.

A detailed description of the implementation of IPC semaphores and messages queues in Kerrighed is available in [**?**].

During the past 12 months, we have designed and implemented checkpoint/restart for IPC resources. The following sections explain how checkpoint/restart for IPC objects occurs from a user point of view implementation of checkpoint/restart for IPC resources and the specifics for each IPC resource type.

2.1 Standard Checkpoint Scenario with IPC Resources

IPC resources are persistent and are not strongly linked with application processes. In this respect, they are quite similar to files, but do not provide a copy mechanism by default.

At checkpoint time, it is impossible to automatically discover which IPC resources have been, and will be, used by the application the user want to checkpoint. Moreover, one IPC resource may be used to help collaboration between several applications. Since the Kerrighed checkpointer is application-based, the user has to choose which IPC resources to checkpoint when they checkpoint an application. That leads to the following scenario at checkpoint:

1. Freeze Application processes
2. Checkpoint the required IPC resources
3. Copy the files in use by the application
4. Checkpoint and unfreeze the application processes

At restart time, IPC resources must be restored before application processes. It is required to ensure that application processes can use the IPC resources as soon as processes are alive. That leads to the following scenario at restart:

1. Restore the IPC resources
2. Restart the processes

Example Figure 1 illustrates an application started from a shell composed of 2 threads using one message queue and writing data to files *error.log* and */dev/pts/1*.

To start the application, the user has typed:

```
$ krgcr-run ./myapp 2> error.log
```

The following commands will be used to checkpoint the application and the content of the message queue:

```
# Freeze application 6
$ checkpoint --freeze 6
```

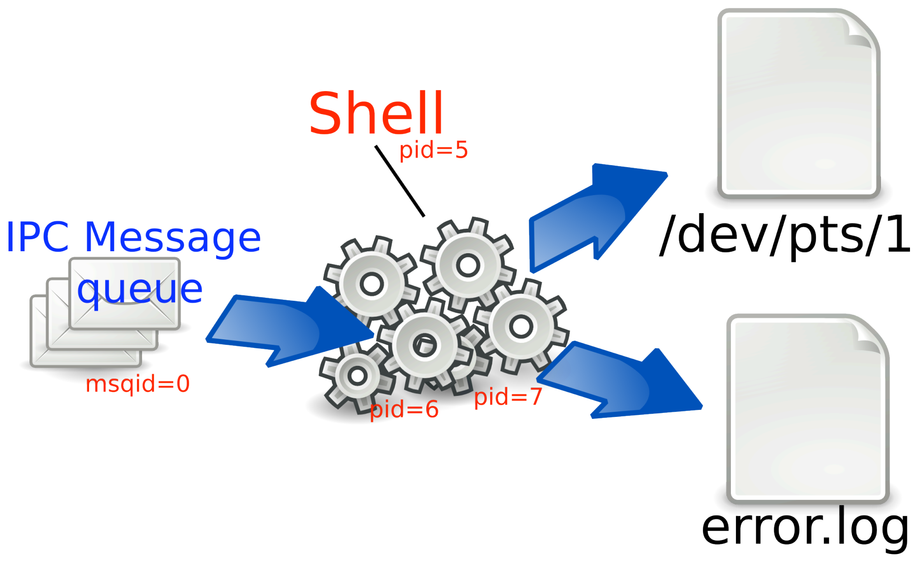


Figure 1: Application composed of two threads using one message queue.

```
# Checkpoint the message queue 0 in dest/msq.dat
$ ipccheckpoint -m 0 dest/msq.dat

# Copy the file error.log that is needed by the application
$ cp error.log dest/

# Checkpoint application 6
$ checkpoint -c 6

# Unfreeze application 6
$ checkpoint --unfreeze 6
```

with `6` being the pid of one process of the application and `0` the identifier of the message queue. File `/dev/pts/1` is not copied since it is a virtual file corresponding to the terminal.

The application can be restarted with the following commands:

```
# Copy back the file error.log
$ cp dest/error.log error.log

# Restore the message queue 0 from its snapshot
$ ipcrestart -m dest/msq.dat
```

```
# Restart application 6 from snapshot version 1
$ restart 6 1
```

with `1` referring to the *first* checkpoint of application `6`.

2.2 Saving And Restoration Of An IPC Resource Identifier

Each System V IPC resource can be identified by its *identifier* or by its *key*. A key is an integer chosen by the application programmer. It is traditionally created with the `ftok()` function to limit key collisions with other applications. This key is given to the `msgget()`, `semget()` and `shmget()` functions to create a new IPC resource or to retrieve the identifier of an existing IPC resource. The mechanism managing the mapping between identifier, key and resource is the same for the three kinds of IPC resources.

Once an IPC resource with key *k* and identifier *i1* has been removed, a new IPC resource may be created with the same key *k* and is likely to have a different identifier, such as *i2*.

The list of IPC objects with their identifiers and keys can be visualised with the `ipcs` command.

The mechanism handling the mapping between key, identifier and resource has been extended to allow reservation of a given identifier and a given key atomically. This extension is needed for IPC resources' restoration as they need to have the exact same keys and identifiers as before.

2.3 Checkpoint/restart of Semaphore Arrays

Semaphores are basically counters used to control access to shared resources for multiple processes. The semaphore value is positive if the protected resource is available and 0 if that resource is busy. This simple mechanism helps synchronizing multithread and multiprocess based applications.

System V IPC semaphores are a set of one or more semaphore values that are sometimes called semaphore arrays. Calling `ipccheckpoint` on a semaphore array leads to a dump of the values of each semaphore. Restoration of said semaphore arrays using `ipcrestart` consists in creating new semaphore arrays with the same key and identifier (see section 2.2) and initializing the semaphore values to the saved ones.

System V IPC semaphores provide a fail-safe mechanism for situations in which a process dies without being able to undo the operations that it previously issued on a semaphore. When a process uses this mechanism, the resulting operations are called *undoable semaphore operations*. If the process dies, the values of IPC semaphores are reverted to the values they would have had if the processes had not executed semaphores operations. To revert undoable operations, each process has a link to a `struct semundo`. Each `struct semundo` refers to one System V IPC semaphore array. `struct semundo` are linked per process and per semaphore array.

In Kerrighed, we have chosen to checkpoint `struct semundo` with the related process and not with the related semaphore array. When restarting the process, the per-process list of `struct semundo` will be restored and reattached to the corresponding semaphore arrays. If one of those semaphore arrays no longer exists, the process restoration fails. The semaphore arrays must still exist or must have been restored, to restore a process that has created undoable semaphore operations.

The list of processes blocked on a semaphore operation are not saved, nor restored while saving/restoring a semaphore array. Instead, each restored process redoes the operation as soon as it wakes up.

2.4 Checkpoint/restart of Message Queues

Message queues provide an asynchronous communication protocol, in other words, the sender and receiver of the message are not required to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them.

Checkpointing a message queue consists of saving the content of `struct msg_queue` and the messages. Restoring it consists of creating a new message queue with the same key and identifier and append the saved messages.

The list of processes blocked reading/writing on a message queue are not saved, nor restored while saving/restoring a message queue. Instead, each restored process redoes the operation as soon as it wakes up.

2.5 Checkpoint/restart of Shared Memory Segments

A shared memory segment consists of memory pages that can be mapped into the address space of several processes.

Similar to other IPC resource types, shared memory segments can be checkpointed/restarted independently of processes attached to it. Checkpointing a shared memory segment consists in dumping the content of the related memory pages that have been touched and skipping the other ones.

Process checkpointing has been modified to not checkpoint memory pages that come from a shared memory segment. However, the process restoration has been modified to reattach the shared memory segment for the relevant pages.

The shared memory segment must still exist, or must have been restored, to restore a process previously attached to this shared memory segment. Since several snapshots can exist for a shared memory segment and be stored in various places, a kernel checkpointer cannot automatically decide from which snapshots the segment must be restored.

A restored shared memory segment can, of course, be used by processes that have not been checkpointed.

3 Dynamic streams with checkpoint/restart

In common cases, communicating processes running on a single machine may use protocols such as TCP/IP, UDP/IP, Unix sockets, System V queues or pipes. In a single system image like Kerrighed, we have to support such mechanisms to allow local communication inside a single cluster. The quick, but inefficient way to provide this feature is (*i*) to forward all communication data to a communication end-point, (*ii*) send regular communication and (*iii*) send another communication forward between a second end-point and the other process. The drawback of this method is the large overhead introduced regardless of the actual process location: processes running on the same node, must forward all their messages!

The purpose of this task is to allow efficient network communication between processes running in the same Kerrighed cluster. In our mind, "efficient" means little or residual performance overhead. our aim is avoiding message forwarding between cluster nodes when processes migrate.

As detailed in previous works [**?**, **?**], we propose a generic approach (named Dynamic Streams) in order to provide standard local communication tools (mainly PIPE, SOCKET and CHAR-device) compliant with process enhancements such as process migration and process checkpoint/restart.

3.1 Dynamic Stream overview

Regardless of the protocol, communication between two processes comprises two distinct data types:

- a binary stream from node A to node B;
- a set of meta-data providing information about the state of the stream and how to handle it (i.e. the protocol implementation itself).

The Dynamic Stream architecture is based on this split:

- avoid any changes retaining the raw performance of the communication tools (IO-mode);
- keep track of any meta-data to be able to extract and introduce the state of communication tools (MD-mode);

Finally, a Dynamic Stream allows only two modes. The first one, (IO-mode) allows only *send* and *receive* operations. In this mode, there may be some protocol management operations. The second mode (MD-Mode) is dedicated to protocol state management, in this mode you cannot have *send / receive* operations at all (i.e. all communication data are in a defined and static state). When a stream is in MD-Mode, all regular protocol operations may occur, as well as, enhanced ones.

3.2 Dynamic Stream framework

The concept of dynamic streams, on which standard communication interfaces are built, has been proposed. We call the extremities of these streams "DynStream sockets" and these can be migrated inside the cluster. Dynamic streams and DynStream sockets are implemented on top of KrgRPC, the point-to-point communication service provided by Kerrighed. By design, in Kerrighed, most of the network protocol properties (such as reliability and message order) are provided by KrgRPC.

As part of the Dynamic Streams framework, there are some specialized interfaces (one per targeted protocol). We use Dynamic Streams, implemented by the DynStream layer to offer dynamic versions of standard Unix stream interfaces (mainly INET / UNIX sockets and PIPE). The DynStream layer implements the abstraction of dynamic stream and DynStream sockets. It is a distributed service which provides global stream management cluster wide.

As part of Task 2.2.11, we primarily focus on design and implementation of the DynStream layer and the TCP/IP socket interface.

3.2.1 Dynamic Stream Service

We define a dynamic stream as an abstract stream with two or more defined DynStream sockets and with no node specified. When needed, a DynStream socket is temporarily attached to a node. When two DynStream sockets are attached, *send / receive* operations can occur.

A dynamic stream is principally defined by several parameters:

- Number of sockets: This specifies the total number of available sockets for the stream. Depending on the stream type, this value may increase, decrease or be constant.
- Number of connected sockets: It specifies the current number of attached sockets.
- Data Filter: It allows modification of all data transmitted with the stream permitting, for example, cryptography, backup or other tasks.

3.2.2 Dynamic Stream Socket

The DynStream service provides a simple interface to allow upper software layers implementing a standard communication interface to manage DynStream sockets[1]:

- *create* / *destroy* a stream;
- *attach*: get an available DynStream socket (if possible);
- *suspend*: detach temporarily a socket, then assign it a handle to reclaim the DynStream socket later;
- *wakeup*: reactivate a previously suspended DynStream socket;
- *unattach*: release an attached DynStream socket;
- *wait*: wait for the stream to be in a clean state (no pending data prohibiting process checkpoint or migration).

DynStream provides two other functions (*send*, *recv*) for I/ O operations.

The dynamic stream service manages (*i*) allocation of DynStream sockets when needed and (*ii*) keeping track of these DynStream sockets through a socket map. When the state of one DynStream socket changes, the stream's manager takes part in this change and updates all other DynStream sockets related to the stream. With this mechanism, each DynStream socket has got the hardware address of each corresponding socket's node in the map. This way, two sockets can always communicate in the most efficient way (using this map).

At the end of a connection, processes are detached from the stream and the stream may be closed.

As with most other Kerrighed distributed services, DynStream relies on the KDDM service in order to manage its data. Currently, DynStream service uses 3 different KDDM sets:

- one dedicated to the stream meta-data: mainly ID, state, number of, and list of end points;
- one dedicated to socket meta-data: mainly ID, state, stream ID and map ID;
- one dedicated to maps management: mainly ID and the map.

[1]for the sake of clarity, names are model function names and not real API names.

3.2.3 Checkpoint/Restart model

A dynamic stream is a self-contained system object which may be checkpointed in an autonomous manner. To checkpoint a DynStream you have to checkpoint all KDDM objects related to this stream from the stream ID and following the logical links to the sockets. Maps don't need to be checkpointed because we have to build them anyway.

The *wait* primitive from DynStream's interface ensures the stream is in a safe state:

- no ongoing socket (i.e. attached process) migration;
- no other ongoing checkpointing operations on the same stream;
- no waiting data in the stream.

This last point is a current prototype limitation but it may be removed in the future.

3.3 Example of Dynamic Stream implementation in a standard communication protocol

Owing to the split design of Dynamic Stream with only point-to-point IO management, nor protocol communication management, it is simple to add Dynamic Stream features to an existing network protocol.

To support Dynamic Stream in a protocol like TCP/IP, we simply create a Dynamic Stream every time the system creates a loop-back connection (i.e. on any local IP address). In doing so, we can have, simultaneously, a process using the regular TCP/IP network stack communicating with a peer outside the cluster, while the same process transparently uses (from the user-space point of view) Dynamic Stream in order to communicate with a peer inside it.

Porting TCP/IP on top of the Dynamic Stream framework requires:

- to modify the *accept*/*connect* protocol methods in order to create (when needed) the dynamic stream;
- to modify the *send*/*recv* protocol methods in order to use the dynamic stream, if available.

As mentioned earlier, Dynamic Stream is an IO management framework, therefore Dynamic stream provides no particular support to address space management like IP address and TCP port number. To provide full TCP/IP support, we created a dedicated KDDM set tracking the state of TCP ports in the cluster. As a complement to Dynamic Stream, this KDDM set provides the cluster-wide protocol view and state management of TCP ports.

4 Outgoing IP communication

The purpose of Task 2.2.10 is to provide a support for TCP/IP communication between a process running inside a Kerrighed cluster and another running outside the cluster. Task 2.2.11 "Dynamic Stream" shares the same goal, but with an additional assumption: both processes are running inside the same Kerrighed cluster. In that case, and in order to have good performance, we only to preserve the system API, but we make some modifications on the communication protocol itself.

In the Task 2.2.10, the main goal is to support in a fully transparent way network communication (mainly IP/TCP or IP/UDP) between a process A running inside a Kerrighed cluster and a process B running outside the cluster. In this task, we may migrate, distant fork or checkpoint the process A without any protocol issue with the process B. For instance, the process B must not detect that the process migration of A from N1 to N2 (inside the same Kerrighed cluster).

In the Task 2.2.10 we allow performance overhead as long as we provide a full transparent mechanism at the protocol layer. In the next Task 2.2.11, we avoid any performance overhead but we allow major (and not transparent) protocol changes.

In the first part of this section, we will describe the key-idea of our frameworki: Distributed IP. This framework will be describe in the second part of this section and we will provide some element regarding checkpoint/restart operation related to our framework.

4.1 Distributed IP address overview

By design, TCP/IP is a connected protocol that manages some end points, also called *sockets*. Depending on their usage, we distinguish two kinds of sockets: *protocol*-sockets and *io*-sockets. Roughly speaking, in a first approach we could say:

- *send* and *recv* operations are related to *io*-sockets;
- all operations but *send* / *recv* are related to *protocol*-sockets.

One major exception is the *select* operation that can be considered in both cases.

For instance, if a server needs to wait for a new connection, it needs a *protocol*-socket. When a request for a new connection arrives, a new socket is created from the waiting one. This new socket is dedicated to define the new stream and this socket socket will be mainly used as an *io*-socket. In our mind, a socket may switch between the *protocol*-socket state and the *io*-socket state several time.

From this, we decided to privilege performance and transparency for ongoing connections (i.e. *io*-socket, the most common case) and to privilege flexibility and management elsewhere (mainly *protocol*-socket, quite unfrequent case compared to *send*/*recv* operations). The retained architecture allows *protocol*-socket migration and manages (if needed) remotely the *io*-sockets. The point is to assume that it is preferable to migrate *protocol*-socket (and future connections) than *io*-socket.

4.2 Distributed IP address framework

Having the capability to migrate protocol-sockets means that connection requests must be handled by the right node. In our opinion, there are only two ways do to that: (*i*) either all requests are handled by one single node and then, eventually, dispatched to the right one; (*ii*) either each request is directly managed by the right node. For scalability reasons, the later is better: since we cannot change the TCP/IP protocol (we cannot make any assumptions about the client) we have to manage the same IP address on several Kerrighed nodes.

The regular Linux kernel provides the CLUSTERIP framework in order to allow several nodes to manage the same IP address. The basic idea of CLUSTERIP is to coordinate a set of nodes in order to allow each node to receive IP frames regarding a static distribution based on properties like:

- *source IP*;

- *source IP*, *source port*;
- *source IP*, *source port*, *destination port*;

Using a virtual broadcast Ethernet address bound to the distributed IP address, all nodes may receive all packets sent to it. Furthermore, regarding a given key, the same node will always manage all IP frames matching this key.

The integration of the CLUSTERIP framework into the Kerrighed will provide an intelligent means, without requiring additional software or hardware, to:

- split the global incoming stream on certain criteria;
- split the global outgoing stream by *protocol*-socket migration.

4.2.1 Protocol-socket management

The most important part of protocol-socket management is providing a global and coherent view of the TCP port address space (*i*) from user-space and (*ii*) from the CLUSTERIP framework. Major properties of such sockets include:

- linked IP address;
- ID of the node receiver (i.e. the ID of the node on which the receiving process is running);
- protocol state of the port.

As usual in Kerrighed, we created a KDDM set in order to manage and share this address space cluster-wide. Since processes may migrate from node to node, the "node receiving ID" may change over time. We replaced the locate function provided by CLUSTERIP by one based on this KDDM set. Thanks to Kerrighed, CLUSTERIP can now use a dynamic distribution.

4.2.2 IO-socket management

In this architecture, IO-sockets are not supposed to move over time, but since processes are allowed to migrate, we need a mechanism to keep them linked to their sockets. Thus, we enhanced the FAF-mechanism (File Access Forwarding) already available in Kerrighed.

4.2.3 Checkpoint/Restart model

With the current version we decided to focus the checkpoint/restart mechanism on protocol-socket management. In this case, we can checkpoint applications using sockets in the ACCEPT-state. When a process checkpoint occurs, we need to save the related KDDM object as part of the checkpoint.

5 Customizable scheduler

During the course of the project we have developed two components: the Pluggable Probes and Policies (PlugProPol) framework and the job submission system that complies to the DRMAA specifications [**?**]. The components are described in three former deliverables: D2.2.6 [**?**], D2.2.7 [**?**] and D2.2.8 [**?**]. Since both components were already stable at the time of the deliverable-writing, all the deliverables are still relevant. Thus, the reader should read those to get in-depth details about the components.

In the last period of the project, the main goal concerning the customizable LinuxSSI scheduler area has been the implementation of Scheduling Configuration Changer. This is the framework for reconfiguring the probes and scheduling policies based on the current cluster load. An example of its capabilities would be the means by which the system can automatically extend the probing period, reducing overhead due to instrumentation of the cluster. When the system load returns to a more normal state, the probing periods are adapted again. Thus, all decisions about scheduling strategies and about probing can be based on system parameters like the overall load, similar to the mechanisms described in [**?**, **?**].

These mechanisms add the capability to dynamically adjust varying computational demand. Furthermore,this scheduler exchange mechanism can be used to prioritize different goals at different times. i.e. A well thought out policy would migrate several interactive jobs that are idle during the night to a small subset of machines and switch off all other machines. This may lead to larger energy savings which impact a project's budget.

Additionally, the described scheduling strategy can adapt itself to the load of the cluster. That is, the system can automatically extend the probing period in order to reduce the overhead due to the measurements. When the load of the system is back to the normal state, the probing periods are adapted again. Thus, all decisions about scheduling strategies and about probing can be based on system parameters like the overall load, similar to the mechanisms described by Franke et. al [**?**, **?**].

5.1 Terminology

Before we start describing the Scheduling Configuration Changer, we should define the basic concepts used within the description. A more detailed description of the entities below can be found in [**?**]:

Probe: an entity for measuring different resource properties (e.g. CPU load, CPU speed, total memory, free memory). A probe collects information (data or events) and makes them available to other scheduler components. Scheduler designers can implement these probes as separate Linux kernel modules and insert them dynamically into the kernel. Probes may collect data already computed by the kernel (CPU average load for instance) or compute other data (alternative measures of CPU load, like one used by Mosix [**?**]), resulting in a set of resource properties being measured.

Scheduling policy: an implementation of a job scheduling algorithm. In Kerrighed, a scheduling policy is in charge of selecting a proper node for a particular process. Scheduling policies base their decisions on data/events collected by probes. An example of a scheduling policy is a migration-based, sender-initiated load balancing policy. This policy takes resource properties from one or more probes as input and when it detects that the local load is higher than the loads of remote nodes, it attempts to migrate processes balancing the load.

Filter: an intermediate entity taking data/events as input and producing a filtered output. Filters are useful to share probes between several scheduling policies and adapt the probes' outputs to each scheduling policy, or to implement modular features of a scheduler. Filters can be chained to implement complex filters out of simple ones. Filters implement different caching policies of data collected from remote nodes, or block events unless the value of the event source exceeds a threshold.

Scheduling configuration: a set of probes, scheduling policies and filters that is loaded on a cluster node at a particular moment. Only one scheduling configuration can be selected at a given moment. Different scheduling configurations correspond to different cluster states (the cluster states are defined using various resource measurements). For example, we can define two separate scheduling configurations: one is selected when the cluster load is low (e.g. the average CPU usage is smaller than 50%) and the other is selected when the load is high (e.g. the average CPU usage is greater than 50%).

5.2 Scheduling Configuration Changer

Scheduling Configuration Changer (SCC) is a component that manages the choice of an appropriate scheduling configuration (see the definition of the scheduling configuration in Section 5.1) based on different resource measurements (e.g. CPU load, ...). Selection of a scheduling configuration is completely automated. The user creates a customized settings file in which they define the scheduling configurations. It also implements the selector used for choosing the appropriate scheduling configuration (see the definition of selector in Section 5.1). Then the settings file is loaded into SCC. SCC then manages loading, unloading and parameter changes for all probes, scheduling policies and filters. It also manages component connections. Since no user intervention is required, the scheduler is more capable of adjusting to the dynamic state of a cluster.

The scheduling policies loading, unloading and parameter changing is done by using the "Pluggable Probes and Scheduling Policies" framework that we implemented in the beginning of the XtreemOS project. This is described in detail in [**?**]. This simplifies the design of SCC considerably, since the component only has to implement the parsing of user-defined scheduling scheduling configuration files and logic for triggering the loading/unloading/parameter adjustments.

SCC runs in user-space provides for a more sophisticated implementation providing:

- A more robust and user-friendly interface: since SCC runs in user-space, we can implement settings files as XML files. Users can read and modify such files easily. If SCC was implemented in kernel-space its configuration would be much more complicated (we would probably have to use prcfs pseudo-filesystem).

- A better working scheduling configuration selector: in user-space, we can implement more complex logic for selecting scheduling configurations. One can use computational intelligence libraries for deciding which configuration is ideal, see Franke et. al [**?**]. In kernel-space, it would be impossible to use user-space libraries. Consequently, the selection logic would have been much simpler and less efficient.

The disadvantage of running SCC in user space is overhead induced due to system calls required for SCC to communicate with PlugProPol residing in kernel space. We claim this overhead to be

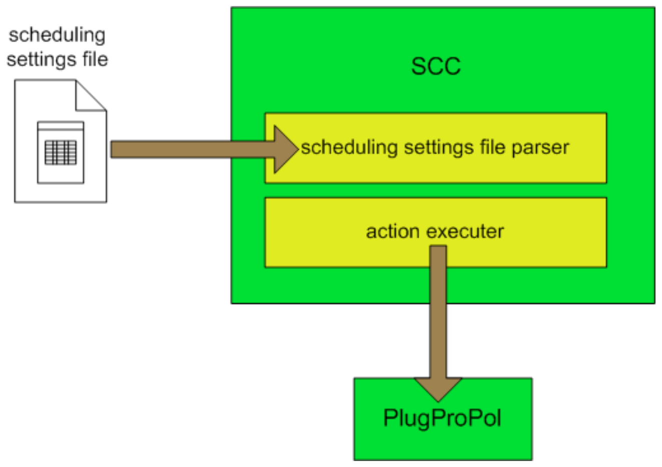


Figure 2: Architecture of Scheduling Configuration Changer.

insignificant, since the communication with PlugProPol is infrequent, occurring only once every few seconds.

The SCC component consists of two main parts (see Figure 2). The first part is the **settings file parser**. It is used for parsing the scheduler's settings files and extracting all required data for scheduling configurations. We decided to write scheduler settings in XML format. This format can be read and modified very easily by the users and can easily be parsed by programs. The structure of the settings file is presented in Figure 3. This file contains definitions of all available scheduling configurations. Each scheduling configuration contains a list of actions PlugProPol framework has to perform when a given scheduling configuration is selected: loading and unloading of probes/policies/filters, setting their parameters (e.g. for probes, we can set their probing frequency, for scheduling policies, we can set the thresholds which trigger process migrations), connecting scheduling policies to probes, etc. The settings file also contains a path to the selector, a component which initiates the exchange of scheduling configurations. The selector is implemented by the user and contains all the necessary logic for selecting an appropriate scheduling configuration based on different resource measurements.

The second part is the **action executer**. It initiates commands to the PlugProPol system to load and unload probes/policies/filters, change their parameters, make connections, etc. These actions are performed every time the scheduling configuration is updated. This component also reads resource measurements from the probes and passes them to the selector.

A single SCC instance manages only the node where it is running. As a consequence, each cluster node needs to host its own SCC instance. We have chosen this per-node approach since it allows us to provide greater efficiency in scheduling strategies than an approach with one centralized SCC instance. At any given moment, scheduling configurations on different nodes can vary. Each node can adapt the scheduling configuration to its local load. Inherently, this type scheduling has greater adaptation capabilities than a single global scheduling configuration based on an average global state in the cluster.

Consider an SCC system configured such that it adjusts automatically, among other things, the probes' measurement frequency to the CPU usage: when the CPU usage is high, probe measurements are less frequent as opposed to times when the CPU usage is low. Thus, the SCC system reduces

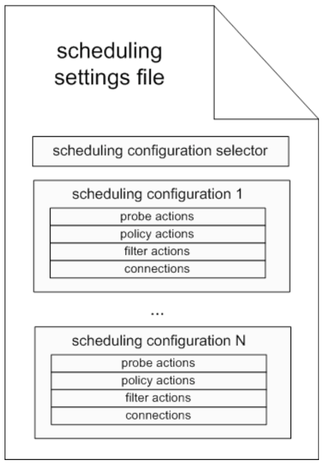


Figure 3: Structure of the scheduling settings file.

probing overhead when more processing power is requested. Having a separate SCC service on each cluster node is a more optimal solution by tuning the measurement frequency of a single node versus a global SCC service with a common probing frequency for the whole cluster. That said, per-node scheduling also has some disadvantages. Multiple SCC services need a means to exchange data. This makes the SCC system more complex. Furthermore, it is possible a group of nodes will continuously swap processes among themselves due to varying scheduling configurations on each nodes. Implementing a prevention mechanism to limit this behaviour is part of our future work.

For a user to establish a customizable scheduler in the cluster, one must create a custom settings file containing the scheduling configurations for different states of the cluster. Each scheduling configuration needs to contain the list of probes, policies and filters necessary to be loaded when the configuration is selected. The custom file must include the list of all connections between PlugProPol entities. The user must provide the implementations of all necessary PlugProPol entities. He can use existing entities or implement them himself.

One of the most important things when establishing scheduling is implementing a selector. As already mentioned in Section 5.1, the selector implements all necessary logic for selecting a particular scheduling configuration based on different resource measurements. In SCC, the selector is implemented as a simple string condition in which the user describes the cluster state (in terms of the values of the different probe measurements) when a particular scheduling configuration is activated.

6 Pushing SSI mechanisms into Linux mainstream development

This section describes the continuing effort to push LinuxSSI to the Linux community. The motivation for this was to attempt to to move parts of LinuxSSI into the mainstream Linux Kernel. Similarly, we wished to discover a path that would make the ideas and concepts behind LinuxSSI and Kerrighed more visible to the Linux Kernel developer community, along with finding users and use cases outside the LinuxSSI project that could supported integration into the Linux Kernel.

The first step in this process was the push for a generalization of the network handling and the replacement of the initial *netdev* based network support with a more generic model that uses events and allows easy asynchronous programming. This occurred with the switch to TIPC [**?**]. Unfortunately, the implementation still much resembled the old design, containing an additional reliability and packet assembly/reordering layer on top of TIPC.

The second step was described in deliverable D2.2.9 [**?**] on the first prototype of a standalone KDDM module. The achievements of that step were:

- Splitting out the core distributed objects infrastructure from LinuxSSI/Kerrighed, compilable easily as a set of modules depending on each other.
- Improvement of hotplug handling, basing it onto TIPC mechanisms. This allowed for cluster "autostart".
- Elimination of some hard-coded IDs, adding mechanisms to allow dynamic usage of I/O linker and RPC ID, in order to ease the adoption of KDDM for external users.
- Finding use cases and developing example code for the standalone KDDM usage.

The standalone KDDM code is self-contained and requires no patching of kernel code. It consists of a set of components that still have a considerable code size, therefore need some analysis with respect to their chances to "land" in the Linux kernel:

- The communication layer: implemented on top of TIPC, contains a reliability and reordering layer, somewhat similar to mechanisms used in the IP protocol. Trying to push this component alone seems to make Little sense, it is a slight extension of TIPC, but with no obvious uses or users outside Kerrighed.
- The hotplug management layer: reacts to nodes appearing or disappearing from the network. Inside the standalone KDDM component this was implemented with TIPC notifications eliminating the need for kernel patching. Unfortunately, this change has not yet been included in the LinuxSSI / Kerrighed main development.
- The RPC layer: allows for registering services that can be invoked remotely. The particular semantics are used widely throughout LinuxSSI / Kerrighed code, therefore this component is crucial and difficult to replace. Unfortunately, the Linux kernel already has various RPC implementations for different purposes, the most prominent one being *sunrpc*. The KDDM RPC layer is an interesting and useful component when considered together with the lower communications and hotplug layer, as it would allow for simplified distributed application programming. Conversely, it is built on top of TIPC, i.e. it supports only Ethernet based transports. Other RPC implementations in the kernel have similar functionality, and even better management of registered functions (e.g. "portmap" for *sunrpc*).
- The distributed object management layer: implements the distributed object coherency protocol. That is based on MESI [**?**], which is a rather aged concept. This implementation needs a quite complex state machine using multiple acknowledgments in some cases. This component relies heavily on the lower layers and is very interesting for simplifying distributed programming. However for applications like shared distributed caches one expects reliability and good performance through support for high speed interconnects.

- Helper tools: contain things like for example yet another implementation of a hashtable. The kernel already has several of them, certainly yet another one would lead to criticism.

The example applications for KDDM are a distributed file system and a distributed lock manager. Unfortunately, the lack of recovery mechanisms in the case of node failures makes these distributed applications rather unreliable, thus uninteresting. Therefore, the strategy of trying to motivate a push of KDDM to mainstream through its applications seems flawed, because the applications aren't reliable yet.

Alternatives to the approach described above are:

- Fix the KDDM reliability and fault recovery code and bring the entire code into acceptable shape. This is a serious development task and would take many man months. In addition, this development would be decoupled from the LinuxSSI / Kerrighed development, and additional effort for getting the changes accepted in Kerrighed would be expected.

- Start reworking a sub-component of KDDM that might be interesting for other in-kernel applications as well. The natural candidate is the RPC layer with the communications layer below.

Obviously, the first target requires too big an effort, while the second one could make sense and is along the path of the rework, anyway. A new network abstraction and new RPC implementation would need to fill some gap in the Linux kernel and provide some features that are currently missing.

The history of this project and its origins in HPC applications as well as the strive for larger scalability suggest that the new RPC layer should support a high speed interconnect that is able to perform remote DMA operations. Infiniband has become the de-facto standard HPC commodity interconnect and its programming is currently supported by the OFED stack [?]. The API allows programming with RDMA and events and is exposed both to user and kernel space in a very similar way: through the verbs interface. Unfortunately the verbs way of programming is difficult and touches rather directly Infiniband host channel adapter hardware. An abstraction providing remote DMA and asynchronous, callback or event based programming would simplify Infiniband programming considerably.

We investigated several abstractions that implement RDMA-style communication with or without callbacks and support Infiniband and similar interfaces.

GASNET [?] from Berkeley National Labs and UC Berkeley is a communications library developed at the for global address space (P)GAS languages like UPC. It manage the required single sided communications, allows for the registration of communication buffers and callbacks. It supports various network interconnects, among others Infiniband and Ethernet UDP. GASNET offers useful general communication abstractions, but has additional features needed for parallel programming, like global synchronization, communication and computation functions in its extended library. Its implementation is purely aiming at user space.

Portals [?] is a network abstraction designed and implemented at Sandia National Labs. Its features mostly aim at MPI implementations, providing ways to match posted receives with incoming messages. Its infrastructure is event based. The reference implementation shows that the design was aimed at putting parts of the code into user space, kernel space or even onto the network card (NIC). Network card support is rather scarce: there is a TCP NAL (network abstraction layer) for user space, one for kernel space (that is outdated and needs porting forward to newer kernels), and support for the proprietary CRAY Seastar interconnect.

LNET (Lustre Network) has been derived from Portals and is the network abstraction used by the Lustre parallel file system [?]. It supports a variety of NALs in kernel space (Infiniband, Myrinet, Quadrics, Ethernet, Portals) and has an API similar to Portals, but with many customizations for Lustre. The code is very tightly integrated with Lustre and it is not trivial to split out into a standalone component. The Lustre developers are not really eager to push LNET into the Linux mainline because they want to remain able to customize it and change the API at any time for their purposes.

The Buffered Messaging Interface (BMI) is the network abstraction used by the PVFS parallel file system [?]. It supports a multitude of network cards including Infiniband, Ethernet (TCP) and Portals. While appropriate for RDMA data transfer, BMI does not provide an API for programming with events.

QLOGIC's PSM is a network abstraction specially tailored to implement MPI in an easy and scalable way. It provides message matching and reliable connectionless communication with very low latency. Currently PSM is proprietary, but in near future it is expected to be opened and integrated into OFED.

The most interesting network abstraction for RDMA based messaging and event based programming seems to be Portals. It was picked up by a hardware vendor (CRAY) and by the most popular high performance parallel file system (Lustre) who implemented different flavors or the Portals specification. An open source variant supporting Infiniband would certainly find its way into the mainstream.

We implemented an RPC layer on top of Portals, tested with the UTCP NAL of the Portals reference implementation in user space. For KDDM and LinuxSSI / Kerrighed the implementation would need to be moved to kernel space, and the missing hotplug feature would need to be added.

Work has started for implementing an Infiniband NAL, for practical reasons this is currently done in user space. While an Infiniband NAL in user space could be finished during the time of the project, the corresponding kernel NAL would not be finished during the project's time frame.

7 Test, integration and support

As a distributed operating system, Kerrighed – on which LinuxSSI is based – has paid close attention to testing and integration with other XtreemOS components.

7.1 Testing

At first, we aimed at providing a large collection of tests which can be run manually by developers or users. These tests must ensure Kerrighed respects the Linux API and behaviour. Additionally, there cannot be any regressions with Kerrighed specific features.

Then, to run theses tests as many times as possible with as many configurations as possible, we setup an automated platform testbed.

7.1.1 Tests suite

As an extension to the Linux kernel, Kerrighed testing must validate the same features as for an unmodified Linux kernel. We then chose Linux Test Suite as a base for the Kerrighed test suite.

Linux Test Suite has been developed since 2000 as a standard test suite for the Linux kernel. The project is maintained by Silicon Graphics Inc., IBM, Bull OSDL and many other contributors.

The original Linux Test Project today covers various aspects of the operating system, for a total of +3000 tests:

- system calls: functional tests for each system call,
- filesystem and disk I/O: stress tests
- networking: stress tests,
- scheduler,
- threads,
- virtual machine management.

The Linux Test Project provides a dedicated API for extending the set of tests. These tests can be written in ANSI-C or in Bash. Provided functions allow performing test setups and cleanups, log test results and more.

We call KTP the suite of tests which is run before each source commit. This suite of tests is made up of a subset of Linux Test Project plus some Kerrighed specific tests. The following tests from LTP are not used:

- stress tests which need several hours to run,
- tests for features known not to be supported currently (*eg.* virtual machine management)

The checkpoint/restart mechanism is the only Kerrighed feature which needs specific tests. 45 tests have been written in Bash, using the LTP API.

All these tests can be run through the `krgltp-smp.sh` script. This script can be found in the `./tests/ktp/` dir of the Kerrighed archive.

Figure 1 presents the number of tests executed as of February 18th, 2010 with the details of passing and failing tests. Tests marked as “crashed” cause a kernel panic when executed.

	Passed	Failed	Crashed	**Total**
LTP - System calls	843	93	3	939
Checkpoint / restart	33	0	0	33
Total	876	93	3	**972**

Table 1: KTP Suite of Tests as of February 18th, 2010.

As a non-regression test suite, we ensure that each development does not decrease the number of passing tests and whenever possible, increase it.

7.1.2 Testing Platform

7.1.2.1 Increasing Testing Reliability

To increase reliability of tests results, we can change two parameters:

1. number of tests: adding tests from LTP which are not currently run, mainly because of execution time (I/O and network stress tests for instance).
2. Number of testing configurations:
   (a) hardware configuration: network interfaces, processors, memory amount and type, *etc*.
   (b) Kernel configuration: more than 2000 parameters can be set when compiling the kernel. Some of them can have interaction with Kerrighed features behaviour.
   (c) Software environment: for reliable results with Kerrighed we must test it with real applications, not only testing atomic system calls. Such testing requires complex setup and cleanup phases.

Changing one or both of these parameters varies testing time. Testing automation is then required if we want avoiding spending more time testing than developing.

7.1.2.2 Requirements

The testing platform must fill the following requirements:

1. manage a cluster of nodes, with the specifics of a Single System Image like Kerrighed,
2. deploy and, eventually, repair a complete environment on the nodes,
3. manage diskless deployment (*ie* networked filesystem root, typically NFSROOT), which is required for Kerrighed,
4. manage kernel configurations,
5. tests result parsing and presentation.

7.1.2.3 Hardware testing platform

An automated testing platform is hosted at Kerlabs' data center. It is composed of a server and 16 available testing nodes amongst 198 of various hardware configurations which are exposed in figure 2.

7.1.2.4 Software testing platform

The Autotest project[**?**] was started in 2006. It is a framework for fully automated testing. It is designed primarily to test the Linux kernel, though it can be used for testing any kind of software or hardware platform. It's an open-source project under the GPL used and developed by a number of organizations, including Google, IBM and many others. The platform core is written in Python.

While some criteria are not implemented directly by Autotest, it is written in a very extensible way so that it can be adapted to our needs, thanks to dynamic typing and the introspection characteristics of the Python language.

Qty	Model	Processor	Memory	Network
70	Dell Poweredge SC1435	2x Opteron 2216 (dual-core) @2.4GHz	4GB DDR2	Gb Ethernet
66	HP Proliant DL145 G2	2x Opteron 246 (mono-core) @2.0GHz	2GB DDR2	Gb Ethernet + Infiniband 10G
62	Sun Fire v20z	2x Opteron 248 (mono-core) @2.2GHz	2GB DDR2	Gb Ethernet

Table 2: Testing platform hardware.

Typically, to extend the Autotest platform, if a feature is handled by a class named `aclass` in the file `apackage/afile.py`, one can write a class named `site_aclass` in the file `apackage/site_afile.py`. Then, instead of using class `aclass` for this feature, the system will use the class `site_aclass`, subclass of `aclass`. While this mechanism is not available for all classes, contributions to enable it for a particular class are easily accepted by the core developers.

From a functional point of view, the platform is made of several layers which all can be run manually or from the layer above. Firstly, a test is described in a Python file. It can be run with the `autotest` command on the testing node. This test can be installed and run on the testing node from the server node, thanks to the `autoserv` command line tool. If necessary or asked explicitly, `autoserv` can deploy a whole system on the testing node or repair an existing system. Finally, a web interface or the `atest` tool can be used to schedule the test. From the web interface, the user can edit the Python test file or use an already defined template. Various kernels with which to execute the tests can be obtained from the web interface.

In figure 4 the overall structure of Autotest framework is presented.

The Autotest platform provides a packaging system for tests. It is already released with 80 test packages, amongst them LTP (Linux Test Project).

7.1.2.5 Test results

The Autotest platform produces a complete log of the testing process, and not only its final results. Logs include:

- installation of the test itself on the machines,
- installation/repairing/cleanup phases when occurring,
- installation and compilation of a kernel,
- results of the tests themselves.

All these logs are accessible from the web interface. This interface produces also customizable views of results, as shown in figure 5.

Currently, the testing platform is operational for compilation and launch of Kerrighed. Kerrighed's suite of tests must be packaged as an Autotest package.

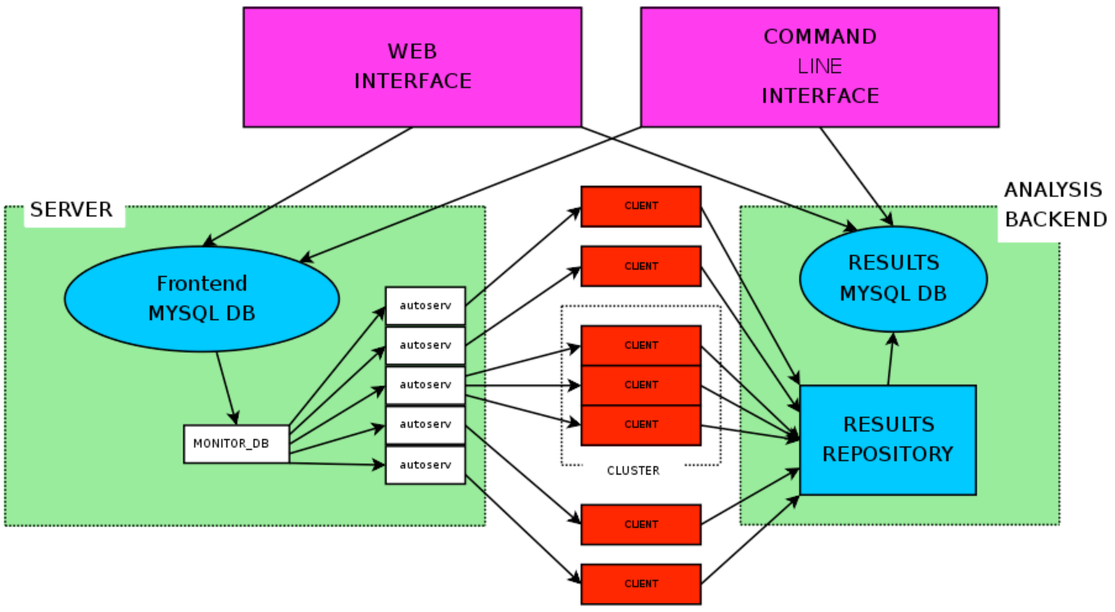


Figure 4: Autotest overall structure.

7.2 Integration

7.2.1 Kernel checkpointer

The Kerrighed kernel checkpointer has been written in a very customizable way. Documentation available as manpages (`krgcr-run(1)`, `checkpoint(1)`, `restart(1)`, `ipccheckpoint(1)`, `ipcrestart(1)`) enabled easy integration of this mechanism with the XtreemOS' GCP service.

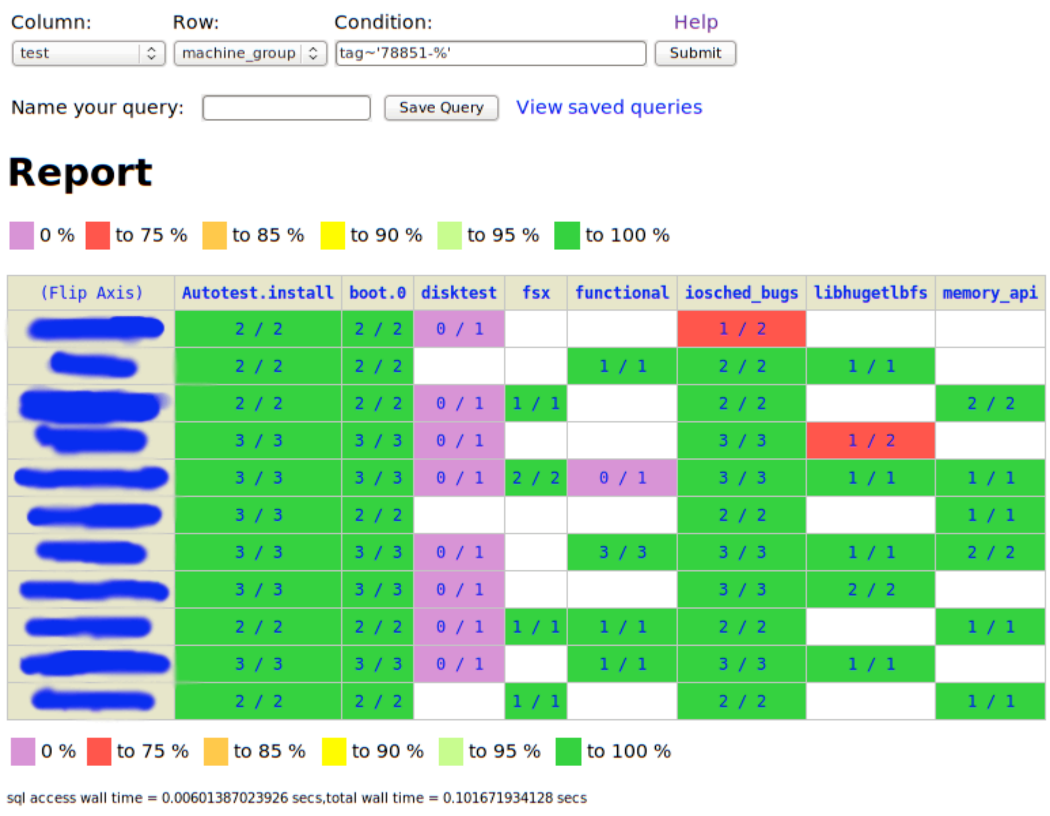
7.2.2 Aladdin - G5K

The Aladdin-G5k project[**?**] (previously Grid5000) is an infrastructure distributed over nine sites around France, for research in large-scale parallel and distributed systems.

Providing users a simple way to deploy LinuxSSI nodes as part of an XtreemOS grid has been done through a tool called xos-ssi-deploy. It deploys LinuxSSI on reserved nodes on Aladdin-G5k.

After deploying a customized image on reserved nodes using kadeploy3, xos-ssi-deploy configures the kernel boot lines while enabling and disabling the appropriate services which should be run on boot. It then reboots the nodes using kareboot and loads the Kerrighed kernel module enabling the nodes to form a cluster. Finally, some configuration directories are mounted through NFS on all nodes followed by a cluster-start sequence.

Such a configured cluster can easily join an XtreemOS grid. In order to achieve that, you must change the appropriate parameters in `/etc/xos/xosautoconfig` and run the xosautoconfig utility. For more info refer to the administrator's guide of XtreemOS.



Column: test | Row: machine_group | Condition: tag~'78851-%' | Help | Submit

Name your query: Save Query View saved queries

Report

🟪 0 % 🟥 to 75 % 🟧 to 85 % 🟨 to 90 % 🟩 to 95 % 🟩 to 100 %

(Flip Axis)	Autotest.install	boot.0	disktest	fsx	functional	iosched_bugs	libhugetlbfs	memory_api
[redacted]	2 / 2	2 / 2	0 / 1			1 / 2		
[redacted]	2 / 2	2 / 2			1 / 1	2 / 2	1 / 1	
[redacted]	2 / 2	2 / 2	0 / 1	1 / 1		2 / 2		2 / 2
[redacted]	3 / 3	3 / 3	0 / 1			3 / 3	1 / 2	
[redacted]	3 / 3	3 / 3	0 / 1	2 / 2	0 / 1	3 / 3	1 / 1	1 / 1
[redacted]	3 / 3	2 / 2				2 / 2		1 / 1
[redacted]	3 / 3	3 / 3	0 / 1		3 / 3	3 / 3	1 / 1	2 / 2
[redacted]	3 / 3	3 / 3	0 / 1			3 / 3	2 / 2	
[redacted]	2 / 2	2 / 2	0 / 1	1 / 1	1 / 1	2 / 2		1 / 1
[redacted]	3 / 3	3 / 3	0 / 1		1 / 1	3 / 3	1 / 1	
[redacted]	2 / 2	2 / 2		1 / 1		2 / 2		1 / 1

🟪 0 % 🟥 to 75 % 🟧 to 85 % 🟨 to 90 % 🟩 to 95 % 🟩 to 100 %

sql access wall time = 0.00601387023926 secs,total wall time = 0.101671934128 secs

Figure 5: Autotest results interface.

7.2.3 Packaging Kerrighed/LinuxSSI by WP4.1

Kerrighed packaging has been designed for easy integration into Linux distributions. Based on the standard autotools suite of tools, it is provided with a `configure` script. With this script, the packager can define all installation paths, with the default values being compliant with LSB, as required in task 4.1.33.

Furthermore, Kerrighed provides an LSB-compliant init script which can be used to start and stop Kerrighed related services.

7.3 Support

7.3.1 Kerrighed community

As many other open-source projects, Kerrighed community support is provided through various channels: mailing lists (`kerrighed.dev@listes.irisa.fr` and `kerrighed.users@listes.irisa.fr`) as well as an IRC channel (\#kerrighed on `irc.freenode.net`).

Furthermore the Kerrighed project website is powered by a wiki engine which allows users to contribute to the documentation and share their experiences.

7.3.2 XtreemOS project

In addition to the channels used for supporting Kerrighed community, support to the XtreemOS project has been given through regular meetings with XtreemOS developers.

A Kerrighed tutorial was given to XtreemOS developers on October 16th, 2009 at INRIA, Rennes, France.

Furthermore Kerrighed developers have participated in weekly audio conferences with other developers, as well as most of face to face technical meetings. Additional collaboration has been done through joint appearances at conferences, along with on site visits of XtreemOS partners at Kerlabs.

8 Conclusion

In this document, the following mechanisms were described: checkpointing of processes which use IPC, the Scheduling Configuration Changer framework for adapting the cluster to current loads, mechanisms for transparent handling of TCP/IP communication inside the LinuxSSI cluster. Additionally, we also described testing and integration activities, along with the pushing of LinuxSSI / Kerrighed to the Linux community.

The work of pushing LinuxSSI / Kerrighed to the mainstream has partially influenced the development of Kerrighed. An analysis of code lead to defining a strategy that aimed at splitting out a reasonably size core infrastructure part and make it a standalone piece of code that can be used generally for distributed programming. The standalone KDDM module was such a piece of code, but a successful move to mainline can only be expected after the rework of some core parts of its infrastructure, especially the communications and RPC layer. Investigations and work have been completed. Ultimately it revealed that the resources and time are not sufficient for finishing a rework in time.

The lesson learned is that pushing code to the Linux kernel mainline is a significant effort that requires full attention and involvement of the developers, who should be prepared to rewrite the code or parts of it several times. The “push” can not succeed if (like in our case) the code was not originally written with integration into the Linux kernel in mind. The code is more driven by commercial needs, while only small parts of it are being reworked towards inclusion.

Since the XtreemOS project is coming to an end, there is not much time left for future work. For the remainder of the project, we will mainly be working on existing functionality, while fixing known issues which are present in the current implementation.